Title: AOT vs JIT: Impact of Profile Data on Code Quality

By

Tyler Wade

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Prasad Kulkarni, Chairperson

Committee members

Perry Alexander

Heechul Yun

Date defended: March 3, 2017

The Thesis Committee for Tyler Wade certifies
that this is the approved version of the following thesis :

Title: AOT vs JIT: Impact of Profile Data on Code Quality

Prasad Kulkarni, Chairperson

Date approved: _____

Abstract

Dynamic or just-in-time (JIT) compilation can generate optimized native code during each program run, and is a popular technique used by process-level virtual machines (VM) to simultaneously achieve binary code portability and high execution performance. JIT compilers typically collect *profile* information at run-time to guide selective compilation and profile-guided optimizations (PGO). Ahead-of-time (AOT) compilation is an alternative model that can also achieve portability by converting the distributed binary to native code when the software is first installed. The AOT compilation model removes the overhead of online profile collection and dynamic compilation, and may reduce VM complexity. However, AOT compilation cannot customize the generated native code to different program inputs/behaviors. The goal of this work is to investigate and quantify the implications of the AOT compilation model on the quality of the generated native code for current VMs.

To achieve our goal, this study develops and conducts many new innovative experiments using a variety of novel frameworks. First, we quantify the quality of native code generated by the two compilation models for a state-of-the-art (HotSpot) Java VM. Second, we determine how the *amount* of profile data collected affects the quality of generated code. Third, we develop a mechanism to determine the similarity or *representative-ness* of different profile data for a given program run for guiding PGOs. Fourth, we investigate how the accuracy of profile data affects its ability to effectively guide PGOs. Finally, we categorize the profile data types in our VM and explore the contribution of each such category to performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Managed high-level languages, such as Java and C#, were designed to be portable. Programs written in these languages are distributed in a *machine-independent* binary format that is designed to ease program execution in a virtual machine (VM) running on many different processor architecture and operating system configurations. Program emulation in these virtual execution environments can be performed by *interpretation* or *binary/native* code execution. Since software interpretation is inherently slow, most performance conscious systems compile (portions of) the distributed code on the target machine prior to execution. Thus, current models ensure program portability while also providing sand-boxed and high performance code execution.

Code compilation in such environments can occur at load-time or run-time. Load-time compilation happens when the program is first installed on the device, and is also called *ahead-of-time* (AOT) compilation. The latest Android runtime (ART) is a recent example of a system that employs AOT compilation [Android Open Source Project, 2017]. Execution-time or dynamic or *just-in-time* (JIT) compilation typically occurs during (every) program execution, and may compile all or only the frequently executed (or *hot*) sections of the program. Many Java-based virtual machines, like Oracle's HotSpot, employ JIT compilation [Paleczny et al., 2001].

While both schemes maintain code portability, the different compilation models have some distinct advantages and tradeoffs. AOT compilation based systems provide several benefits. Most prominently, the compilation in such systems is conducted offline and happens only once, rather than during each program execution. This property eliminates the online compilation overhead, and is particularly beneficial to short-running programs that have a relatively flat method hotness curve. Additionally, AOT based systems may also reduce the VM complexity by not needing the

selective compilation, code cache, and related runtime infrastructure. On the other hand, since JIT compilation happens during every program execution, JIT compilation based system need this supporting VM infrastructure.

The JIT compilation model also has some unique advantages. For instance, JIT compilation based VMs can collect *profiling* data during every program run. Program profiling refers to the set of techniques that can monitor program execution to discover, understand and reason about the dynamic or run-time behavior of a program. The JIT compiler can then utilize this profiling data during profile-guided code optimizations (PGO) to customize the native code for each specific input and improve overall program performance. Such infrastructure also enables the VM to apply additional optimizations *speculatively*. Speculatively compiled code can be de-compiled if the speculative condition is invalidated later.

Google Android, one of the most popular Java-based platforms, recently replaced its JIT based Java VM (Dalvik) with an AOT based runtime (ART) [Android Open Source Project, 2017]. Researchers have also been exploring the use of AOT compilation systems [Jung et al., 2014, Oh et al., 2015, Wang et al., 2011, Hong et al., 2007] to reduce startup time for Java applications. Despite these recent developments, the performance implications of AOT vs. JIT compilation systems are still not clear. This work aims to address this issue by shedding light on how design tradeoffs in managed language compilation systems impact code quality and program performance.

The goals of this research are to: (a) investigate, quantify, and highlight the role of profile data and dependent PGOs to improve generated code quality in managed language runtimes, and (b) understand the challenges that AOT based systems face to generate high-quality code without access to customized and accurate profile data for each program input. We develop a variety of innovative experiments and VM frameworks to resolve the following issues.

1. How does the amount of custom (from the same program run) profile data affect effectiveness of PGOs?

2. How do the inaccuracies in profile data affect the quality of generated code? To quantify the impact of inaccurate *offline* profile data, we develop techniques to systematically introduce

noise in the profile data to study this question. We also develop a mechanism to calculate the *representative-ness* of pairs of profile data sets.

3. What kinds of profile information are most important to performance in current VMs? We study the different types of profile data collected by the HotSpot VM and isolate their individual impact to assess this issue.

We make the following findings in this work limited to the VM (HotSpot) and benchmarks (DaCapo) used.

1. Availability of custom profile information helps the compiler generate significantly higher-quality code.

2. Beyond a small amount of accurate profile data, additional profile information does not derive any further performance benefits from PGOs.

3. Offline profiling works by employing a *representative* program input(s) to gather profiling data and guide PGOs. The generated optimized binary can later be invoked on different inputs. We find that the similarity (or *representative-ness*) between the inputs used during the profile run and later measurement run directly and significantly impacts the quality of code generated by PGOs.

4. The collected profile information is used by the JIT compiler to guide many optimization decisions. Making only a small percentage of such decisions incorrectly significantly reduces the quality of generated code.

5. The runtime can profile the execution to collect many different types of program behavior knowledge. We find that only a small subset of the collected profile data types produces most performance gains in current VMs.

We believe that our research provides greater insight in the workings, characteristics, and benefits of existing profiling based VM optimization systems, and demonstrates some of the challenges that AOT compilation systems must overcome to achieve comparable code quality to JIT based VMs.

Chapter 2

Background and Related Work

In this section we describe some applications of profile data to individual optimization problems, and measured performance benefits. We also present prior work investigating properties of profiling and PGOs, and compare the goals of our current research with related past studies.

One of the first attempts to gather run-time program information to optimize program performance was made at IBM [Apple, 1965]. Their goal was to record the hardware instruction trace of a program to find program hotspots to minimize optimization effort. The use of the term *profile* data was first recorded by Knuth in their study to analyze and exploit properties of FORTRAN programs [Knuth, 1971]. While Knuth's paper defined the term *profile* data to be execution-time statement frequency counts, the term now includes any program behavior information collected at run-time.

Profiling data can be collected using *offline* and *online* schemes. Offline profiling uses additional prior runs of the program to generate profile data. A later compilation can than use this profile to guide code optimization decisions. Offline profiling is used by static compilers like GNU gcc/g++ [Hwu et al., 1993, Mock et al., 2000, Chang et al., 1991, Pettis & Hansen, 1990]. Dynamic or online profiling collects profile information during the same program run, and is commonly employed by advanced managed language run-times, like those for Java [Paleczny et al., 2001, Arnold et al., 2011, Suganuma et al., 2005, Cierniak et al., 2000]. Researchers have also developed static analysis techniques to estimate some run-time information for PGOs [Wu & Larus, 1994]. While JIT compilers typically use online profiling, AOT compilers may employ offline profiling data or static analysis to guide dependent optimization decisions. Some of our studies in this work assess the impact of imprecise profile-based guidance on the quality of code generated by PGOs.

Profile data has traditionally been employed to find the *hot* or frequently executed program blocks or functions. Knowledge of hot program regions can then be used to focus compilation and optimization effort. For example, many Java VMs only compile and apply PGOs to the hot program methods to minimize JIT compilation overhead at run-time, in a technique called selective compilation [Hölzle & Ungar, 1996, Paleczny et al., 2001, Krintz et al., 2000, Arnold et al., 2005]. Profile information is also used to direct many other optimization tasks. For instance, profile data was used to randomize/diversify cold code blocks to reduce overhead [Homescu et al., 2013], during profile-guided meta-programming [Bowman et al., 2015], to improve code cache management in JVMs [Robinson et al., 2016], to improve heap data locality in garbage collected runtimes [Huang et al., 2004], to guide object placement in partitioned hot/cold heaps to lower memory energy consumption [Jantz et al., 2015], etc. Our goal in this work is not to generate new or improve existing PGOs, but to determine how inaccuracy in profile data or static analysis based estimators can impact the effectiveness of PGOs.

Several prior studies compare the accuracy and impact of sampling-based profilers on adaptive tasks. The accuracy of any given profile data can be compared directly with the known correct profile, if it is available [Arnold & Grove, 2005, Duesterwald & Bala, 2000, Moseley et al., 2007]. When the correct profile itself either cannot be generated or is not known, researchers have used causality analysis to assess if their profile is *actionable* [Mytkowicz et al., 2010, Rubin et al., 2002]. An *actionable* profile is one that is able to correctly guide the dependent adaptive task and yields the expected outcome. In our current work, rather than evaluate the accuracy of the profiler, we focus on assessing how well the program behavior with different plausible program inputs (with offline profiling) can represent the program execution with the given program input. To our knowledge, this work is the first to conduct a thorough systematic quantification of representative-ness of different profile data and the effect of such dissimilarity on the effectiveness of PGOs in a standard Java VM.

Chapter 3

Tools, Benchmarks, and Experimental Configuration

All our work for this paper was conducted using Oracle's production-grade Java virtual machine (HotSpot) in JDK-9 [Paleczny et al., 2001]. Our experiments use the standard DaCapo Java benchmarks [Blackburn et al., 2006]. The experiments were conducted on a cluster of Intel x86-64 2.4GHz machines running the Fedora Linux operating system. In this section we provide a brief background on the workings and properties of the HotSpot VM and DaCapo benchmarks that is relevant to this work. We also explain some details of our experimental setup.

HotSpot Internals: HotSpot's emulation engine includes a high-performance threaded bytecode interpreter and two distinct JIT compilers. The *client* or *c1* JIT compiler is designed for fast program *startup*. The *c1* compiler is very fast, but applies fewer and simpler compiler optimizations. The *server* or *c2* JIT compiler is slower and applies a broad range of traditional and profile-guided optimizations to generate higher-quality code for fast *steady-state* program performance. The more recent HotSpot releases also include a *tiered* compilation mode, where the hot code is first compiled by *c1*, and only the most important program methods are later compiled by the *c2* compiler. Tiered compilation is the default mode in HotSpot-9. In this research we focus on *code quality* and therefore only use the *c2* compiler for all our experiments.

Program execution in HotSpot begins in the interpreter. The HotSpot interpreter profiles program execution to collect various program behavior statistics, including the *invocation* and loop *back-edge* counts for all program methods. If the sum of the invocation and loop-backedge counts for a method exceeds a fixed threshold, then HotSpot queues that method to be compiled.

Background Compilation: HotSpot employs a technique called *background compilation*, where JIT compilation occurs in separate OS *threads* in parallel with application execution [Krintz et al., 2000]. Background compilation prevents application stalls due to JIT compilation. However, it can also delay method compilation (relative to the application threads) if the compilation queue is backed up; during which time the method running in the interpreter can continue collecting profile data. Therefore, we disable background compilation for most of our experiments to allow more determinism and control over when each method is compiled and the amount of profile data collected prior to compilation.

DaCapo Benchmark Suite: All our experiments employ 10 of the 14 Java DaCapo benchmarks. Four benchmarks, `batik`, `eclipse`, `tradebeans` and `tradesoap` are excluded because they fail to run with the the default HotSpot-9 setup (without any of our updates or modifications).[1] The DaCapo suite provides two distinct inputs for each benchmark program, called *small* and *default*. Some, but not all, DaCapo benchmarks also have access to a distinct *large* input, which we do not employ in this work.

Our experiments attempt to evaluate the quality of code generated by PGOs during JIT compilation by measuring program execution time after all desired compilations are complete. We exploit a mechanism provided by the DaCapo harness that allows a benchmark to be *iterated* multiple times. To achieve determinism most of our experiments restrict the set of methods compiled to those that are detected to be hot and are compiled in the first program iteration. Each run iterates the benchmark 12 times and measures the program run-time during its final iteration.

Table 3.1 describes a few relevant characteristics for each benchmark used in this work. The first column in Table 3.1 gives the benchmark name. The next column reports the average steady-state program run-time with the default HotSpot setup. The final three columns provide the number of methods compiled by each benchmark during its first iteration (startup), at the end of 12 itera-

[1] `batik` and `eclipse` fail due to incompatibilities that were introduced in OpenJDK 8 and have been observed and reported by others [Github, 2014a, Github, 2014b]. `tradebeans` and `tradesoap` witness frequent, but inconsistent failures with the default configuration. We have not fully investigated the cause of the failures, but we believe it is related to issues reported in [Blackburn et al., 2009].

Benchmark	Steady-State run-time (ms)	Methods compiled		
		Startup	Steady	All
avrora	5449.00	363.60	486.50	4151.90
fop	467.50	593.70	1250.10	7459.00
h2	6595.50	853.60	1042.50	5159.40
jython	3036.20	1361.10	1513.10	7469.70
luindex	845.20	258.20	548.70	4004.50
lusearch	1891.20	379.00	435.00	3366.80
pmd	5054.30	968.30	1492.10	6122.40
sunflow	2227.10	317.80	350.20	4717.00
tomcat	6082.70	996.60	2197.40	24803.10
xalan	1571.50	683.30	1401.10	5073.00

Table 3.1: Relevant properties of used benchmarks from the DaCapo Java benchmark suite

tions (steady-state), and by a compiler that compiles all program methods on their first invocation respectively. To account from inherent timing variations during the benchmark runs, all the run-time results in this paper report the (geometric) average and 95% confidence intervals over 10 runs for each benchmark-configuration pair [Georges et al., 2007].

Chapter 4

Constructed Experimental Mechanisms and Frameworks

We implement many new mechanisms in the HotSpot VM to correctly and fairly conduct our experiments for this study. We note that designing and implementing these frameworks in an advanced VM like HotSpot are challenging tasks, and therefore are major contributions of this work. In this section we describe some of these engineered frameworks.

4.0.1 Enable VM to Detect User-Defined Program Execution Points

Ordinarily, the virtual machine does not possess the ability to efficiently detect user-defined program points as they are reached during execution. We found that many of our experiments would benefit from such a VM capability, especially to detect the start/end of individual benchmark *iterations*. Inspired by prior work in the literature, we make a small update to implement this functionality in the VM [Kulkarni, 2011].

We add an empty *VM-indicator* method to the DaCapo harness that starts the next program iteration and statically annotate the method with a special flag. We extend the VM to mark such annotated methods when the classfile is loaded. The HotSpot interpreter efficiently checks for this flag at every method invocation and directs VM control-flow to custom user-defined code if it is encountered during execution.

4.0.2 Import/Export Profile Data

One important contribution of this work is a mechanism that we built in the HotSpot JVM for exporting profiling data recorded during one instance of the VM and importing it during a later

instance. Advanced static compilers that support PGOs, like GCC (*gprof* [Graham et al., 1982]) and LLVM (*llvm-profdata*), possess the ability to collect and dump profile data from one program execution. The compiler can then use this profile data during a later compilation to guide PGOs. However, such frameworks are uncommon for managed language run-times, such as Java VMs, since they typically rely on online profiling. We found that the types of profile data held by a JVM pose some unique challenges to accurately serialize and deserialize the profile data.

We experimented exporting the profiling data a few different points during the execution of the JVM. First we attempted to dump the profiling data for every loaded method when the VM shutdown. We found, however, that when loading the profiling data, we weren't seeing as many methods being compiled that we exected. Unfortunately, Hotspot decays the invocation and backedge counters of a method when it is sent to compile to prevent the interpreter from attempting to enqueue a method for compilation frequently while it in the compiler queue. As a result, those counters in the exported data didn't accurately reflect the state of the method when it was compiled. So, our second attempt was to dump the data for each method just before it is compiled. This gave us more accurate information for methods that are compiled, but failed to collect data for any methods that were not hot and in practice we got worse performance than expected. When compiling a hot method, the compiler frequently inlines methods, even ones which are not themselves hot. As such, the profiling data of these methods can also affect code quality. Our third, and final, approach was to export data at compile time for hot methods and at VM shutdown for cold methods.

For many data types, including counter and boolean values, the serialization/deserialization process is relatively straightforward. However, there are exceptions like the pointers to the VM structures that represent JVM classes. Since pointer values are specific to each execution instance, we abstract such data types by recording the corresponding class name (including package path), in the serialized format. Later during deserialization, we perform a lookup to find a loaded class structure with a matching name.

Looking up a class name requires that class to have previously been loaded by the VM. The

design of the class-loading infrastructure in HotSpot prevents us from loading classes during the deserialization process. Therefore, we delay the deserialization process until all referenced class names in the imported profile file have already been loaded. In order to achieve a reasonable lookup-hit rate, our framework prevents methods from being compiled during the first iteration of the benchmark and performs the deserialization of the profiling data in between the first and second benchmark iterations. Even with this mechanism, there are a few lookup misses. For example, when loading profiling data for the in the "xalan" benchmark, there are about 6000 class name lookups attempted, of which 19 fail. We analyzed some of these misses and found that many of them come from what appear to be dynamically generated classes with semi-random names. Since there are only a few such cases, we did not yet attempt to resolve or predict the class name in such cases.

Another challenge is understanding and serializing profile data structures that vary in layout depending on the bytecodes that make up the method. Specifically, each method in HotSpot maintains a `MethodData` object, which contains an array of structures that hold the profiling information for particular bytecodes in the method. For example, a virtual call bytecode corresponds to a structure that records the receiver types seen at call site.

We store both the type and bytecode index of each profile data structure in addition to the profiling data so that any mismatches during the deserializing process can be detected. We abort in any such cases of a mismatch to avoid corrupting the data structures during the evaluation run. Such a mismatch could occur if the method was modified since the data was collected. It also prevents profiling data collected by a debug or release build from being loaded by the other build configuration, since the layout of the structures differs subtly between debug and release configurations.

We store the serialized profiling data in simple text files formatted as JSON. We chose JSON because it is relatively human readable. The downside to using JSON over a custom format is its verbosity, affecting the speed of reading and writing. Since we focus on program steady-state time, we are able to ignore the cost of importing and populating the profile data structures during our

experiments. An example of the format of the serialized profiling data follows:

```
[
  {
    "klass": "org/apache/xalan/transformer/StackGuard",
    "name": "checkForInfinateLoop",
    "sig": "()V",
    "loader": "org/dacapo/harness/DacapoClassLoader",
    "mdo": {
      "fields": {
        "_nof_decompiles": 0,
        "_nof_overflow_recompiles": 0,
        "_nof_overflow_traps": 0,
        "_trap_hist": "00000000000000000000000000000000000000000000000000",
        "_eflags": 0,
        "_arg_local": 0,
        "_arg_stack": 0,
        "_arg_returned": 0,
        "_creation_mileage": 0,
        "_invocation_counter": 1,
        "_backedge_counter": 1,
        "_invocation_counter_start": 0,
        "_backedge_counter_start": 0,
        "_tenure_traps": 0,
        "_rtm_state": 2,
        "_num_loops": 0,
        "_num_blocks": 0,
        "_would_profile": 0,
        "_data_size": 928,
        "_parameters_type_data_di": -2,
        "_size": 1272
      },
      "data": [
        {
          "type": "VirtualCallData",
          "count": 0,
          "cell_count": 5,
          "header": 262149,
          "receivers": [
            {
              "class": "org/apache/xalan/transformer/TransformerImpl",
              "count": 5000
            },
            {}
          ]
        },
        {
          "type": "BranchData",
          "header": 851975,
          "taken": 5000,
          "not_taken": 0,
          "displacement": 32,
          "cell_count": 3
        },
        // ...
        {
          "type": "JumpData",
          "header": 10616835,
          "taken": 0,
          "displacement": -792,
          "cell_count": 2
        }
      ],
      "extra_data": [
        {},
        {},
        {},
```

```
        {},
        {},
        {},
        {
          "type": "ArgInfoData",
          "header": 9,
          "cell_count": 2,
          "array": [
            0
          ]
        }
      ]
    },
    "mco": {
      "_interpreter_invocation_count": 5001,
      "_interpreter_throwout_count": 0,
      "_number_of_breakpoints": 0,
      "_invocation_counter": 40009,
      "_backedge_counter": 1,
      "_nmethod_age": 2147483647,
      "_invoke_mask": 1016,
      "_backedge_mask": 8184,
      "_rate": 0.0,
      "_prev_time": 0,
      "_highest_comp_level": 0,
      "_highest_osr_comp_level": 0
    }
  },
  // ...
]
```

4.0.3 Control Method Compilation Order

The order in which methods are compiled in the HotSpot VM is known to influence later optimization decisions, especially for method inlining. Thus, configurations that compile an identical set of methods in different orders can generate different compiled native codes and result in different program run-time performance. Therefore, we build a mechanism in the VM to sort and compile the set of hot methods in an external user-defined order. However, a naïve implementation of such a mechanism may delay the compilation of some hot methods if any other methods that precede it in the sorted order have not yet been compiled. This delay in compilation is problematic for our current study since the delayed methods will continue to collect additional profile data, which can affect optimization decisions.

Our mechanism to resolve this issue conducts the experiment in two runs for each benchmark configuration. The first *training* run uses the framework just described to export the profile data for each hot method at the proper point during execution. In the second *evaluation* run, the first

benchmark iteration is completely interpreted and conducts no JIT compilations. The VM uses the VM-indicator mechanism to detect the end of the first iteration. At this point, the VM stalls the application threads, loads the profile data exported by the training run, and then sorts and compiles the set of hot methods in the given order. The application threads are resumed after all compilation is done. Methods sent to be compiled after this point, for example because they were deoptimized and became hot again, are compiled as normal.

4.0.4 Evaluate Representativeness of Profile Data

A *representative* program input produces an execution that is typical for that program and would closely resemble execution time behavior with a majority of other inputs for that program. Offline profiling based systems attempt to find such a representative input to obtain relevant training data to use to guide PGOs. Unfortunately, there are generally no clear guidelines on how to discover such an input, and given an input it is equally unclear if it is indeed representative.

The representative-ness or similarity of some profile data with some other *current* program profile is a factor of the dependent PGO. In this section we present a mechanism to quantify the similarity of any two program profiles with respect to the profiling decisions they induce during PGOs.

Our mechanism to calculate the representative-ness metric extracts and compares the path taken during compilation when the two program profiles are used. We identified 55 *profile-site* locations in HotSpot's c2 compiler where profiling data is used to inform optimization decisions. We insert hooks at all these locations. When a method is compiled, we make a record of which of these locations is visited and in what order. At each hook, we note the name of the current method being compiled (which disambiguates whether this is an inlined method), the current bytecode-index (BCI), and the unique number of the hook location. The record of these profile-site decisions creates a trace of the path the compiler takes as it makes profiling-informed decisions. At VM shutdown, we dump these traces to a file for later analysis. [1] An example of the profiling decision

[1] We experimented with including the profile-data value seen at each profile-site in our trace, but found that exclud-

information for one method follows:

```
[
    "org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;": [
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 1, 35],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 1, 36],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 1, 51],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 4, 50],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 4, 48],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 32],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 35],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 36],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 3],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 4],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 1],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 2],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 5],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 12, 0],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 4, 50],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 4, 48],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 4, 27],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 4, 5],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 24, 51],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 28, 49],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 52, 50],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 52, 48],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 51],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 3],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 4],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 1],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 2],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 36, 5],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 39, 50],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 39, 48],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 44, 3],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 44, 4],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 44, 1],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 44, 2],
        ["org.python.core.PyFastSequenceIter.__iternext__()Lorg/python/core/PyObject;", 44, 5],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 27, 50],
        ["org.python.core.WrappedIterIterator.hasNext()Z", 27, 48],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 4, 50],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 4, 48],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 4, 27],
        ["org.python.core.WrappedIterIterator.getNext()Lorg/python/core/PyObject;", 4, 5]
    ],
    // ...
]
```

Our technique for measuring the similarity of two traces for a given method is inspired by the Unix `diff` utility. Our mechanism calculates the longest-common-subsequence (LCS) of the two traces and divides two times the length of the LCS by the sum of the lengths of the two individual traces. The resulting ratio gives us a percentage measure of similarity. When calculating the LCS, we treat the tuple of the three recorded data values at each profile-site as an atomic unit, analogous to how the `diff` utility treats individual lines as atomic units when calculating a LCS.[2] To create a

ing it improves run-to-run stability and profile similarity comparison.

[2]We used the `difflib` module of the Python standard library to find LCS.

measure of similarity for the entire program, we average the representative-ness measure of every method that was compiled during both VM instances. This average is unweighted, with no regards to the length or the relative hotness of the methods.

Chapter 5

Experiments, Results and Analysis

In this section we describe the results of our experiments that investigate the characteristics of current profiling-based JIT optimization systems in VMs. These results indicate the challenges that AOT compilation systems, whether based on offline profiling or static analysis, face to achieve the performance of JIT compilation systems that have access to accurate profile information from the current run.

5.0.1 Impact of Profiling on Generated Code Quality

Our first set of experiments are designed to quantify the impact of profiling information on generated code quality. Program execution time serves as our metric for measuring code quality. We prepare five distinct HotSpot configurations to compare the behavior and performance of AOT and JIT compilation systems.

AOT-all: This configuration compiles all program methods on their first invocation. We disable profile data collection. Compilation occurs at the end of the first benchmark iteration. A method compilation order cannot be enforced as we do not have any other baseline configuration. The last column in Table 3.1 gives the number of methods compiled by each benchmark in this configuration.

JIT-steady: VMs employing selective compilation, like HotSpot, only compile methods when they are detected to be hot (their invocation+loop-backedge counts exceed 10,000 in HotSpot). This configuration represents the *steady-state* setting for the HotSpot VM. Profiling is enabled. A method compilation order is not enforced and the methods are compiled as they

achieve hotness in their first twelve iterations. The number of methods compiled by this configuration for each benchmark is given by the third column in Table 3.1.

AOT-steady: This configuration restricts the set of methods compiled to those that are compiled by the *JIT-steady* configuration for each benchmark. However, we do not enable profiling for this AOT compilation. All methods are compiled after the first program iteration, and a method compilation ordering is not enforced.

JIT-startup: This configuration is similar to earlier *JIT* setup, but restricts the number of methods compiled to those that get *hot* during the first iteration with HotSpot's default setting. The number of methods compiled by each benchmark is given by the second column in Table 3.1.

AOT-startup: The configuration is similar to *AOT-steady*, but restricts the set of methods compiled to that compiled during *JIT-startup*. Profiling is disabled, and methods are compiled in the order they reach compilation in the *JIT-startup* configuration as described in Section 4.0.3.

In all cases the run-time of the 12^{th} benchmark iteration is reported to ignore compilation overhead and to allow the execution to stabilize. All experiments run the DaCapo programs with their *default* input.

Figure 5.1 compares program performance with the AOT and JIT compilation models. The first bar for each benchmark in Figure 5.1 plots the ratio of the *AOT-all* and *JIT-steady* configurations,

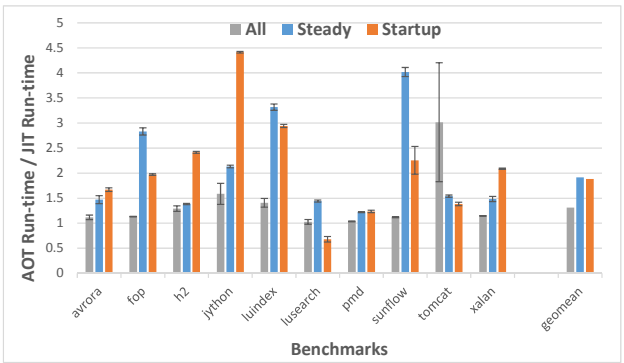

Figure 5.1: Profile data and PGOs have a significant impact on program performance on the HotSpot JVM

the second bar compares the *AOT-steady* and *JIT-steady* configurations, while the last bar compares the *AOT-startup* and *JIT-startup* configurations. The first comparison gives a reasonable estimate of the profiling benefit derived by HotSpot-like VMs that employ selective compilation and may only compile a fraction of the total program methods. The final two plots for each benchmark can be used to estimate the performance gain due to profiling for VMs and benchmarks that have sufficient time and resources to compile all program methods. By enforcing a common method compilation order, the *startup* configurations eliminate one additional source of performance unpredictability, and therefore provide a better baseline for comparison. We use these *startup* configurations in our later experiments.

All comparisons uniformly show that the program profile behavior, which is extensively used by PGOs in current VMs for languages like Java, significantly influences the quality of generated code. AOT compilers may have to rely on mechanisms like offline profiling or static analysis to address this potential loss in performance. However, these alternative mechanisms have other limitations. In later sections we attempt to study the challenges that AOT compilers may need to overcome when using these alternative mechanisms to estimate profile information.

5.0.2 Impact of Profile Data *Amount* on Code Quality

Offline profiling that can be used to drive PGOs in an AOT compiler can collect profile data for the entire duration of the *training* program run, or even over multiple offline runs using different training inputs. In contrast, JIT compilation systems need to balance the amount of profile data collection with the delay in making optimized code available to the emulation engine. Spending too little time understanding and recording the program behavior may have performance implications by incorrectly biasing dependent optimization decisions. Likewise, staying too long in the profile stage will delay JIT compilation, causing the program execution to remain in the inefficient interpreter for a longer duration. In this section we investigate the issue of how much profile data is needed by current PGOs to make correct profile-based decisions and generate the best quality code.

We design a simple experiment that precisely controls the amount of profile data collected during the multiple different training runs. This experiment employs our frameworks described in Sections 4.0.2 and 4.0.3. Thus, the training runs export the collected profile data that is then loaded and used by the evaluation run. We also control the number of methods compiled and their compilation order so that these factors remain uniform across all experimental configurations. We configure the training runs to collect per-method profile information that corresponds to each method executing for 0, 10, 25, 50, 75, 100, 250, 500, 1000, 2500, 5000, 8000, 10000, 25000, and 50000 execution (invocation + loop backedge) counts. By default, HotSpot uses the compile threshold of 10000 for its c2 compiler.

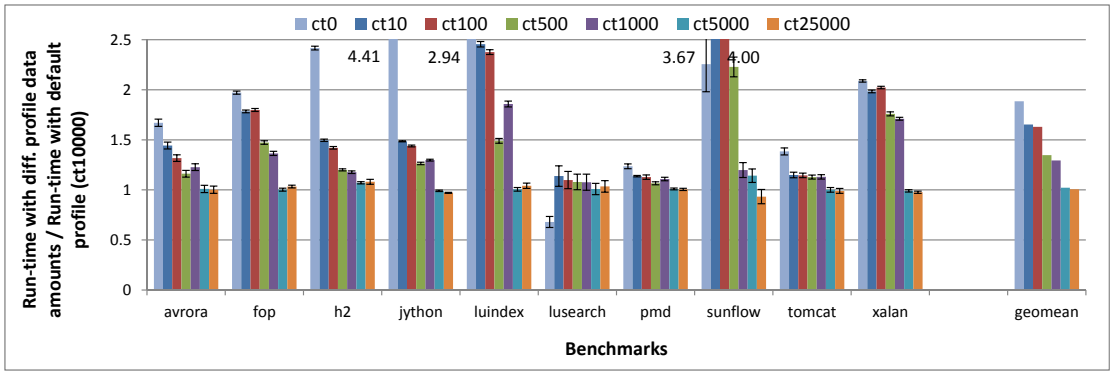

Figure 5.2: A small amount of accurate (from the current program run) profile data is sufficient to effectively guide PGOs on the HotSpot JVM for the DaCapo benchmarks

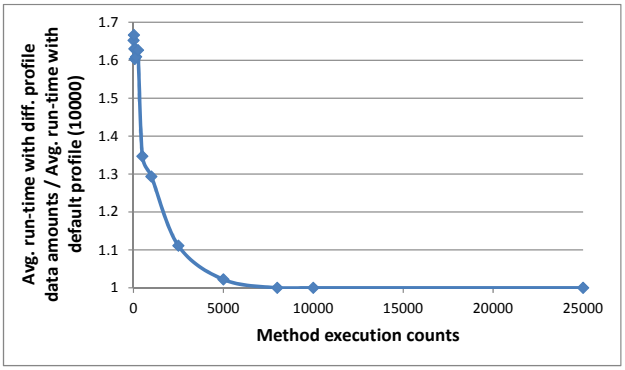

Figure 5.3: Average program performance quickly improves and reaches saturation with small increases to the amount of collected program profile information. Any additional profile data (corresponding to >8000 invocation + backedge counts) did not produce any noticeable performance gain.

Figures 5.2 and 5.3 show the results of this experiment. We find that while some profile infor-

mation is necessary to effectively guide PGOs, the benefit to performance obtained from increasing profile knowledge quickly reaches saturation beyond which we do not notice any significant additional performance gain. Thus, profile data collected over very low method execution counts ($<$ 100) are unable to provide effective guidance to PGOs in HotSpot. Performance increases rapidly with growing profile data with moderate method execution counts ($<$ 5000). We see no significant performance gains with profile data from execution counts beyond 8000 in our experiments with HotSpot and DaCapo. We also notice a situation (for *lusearch*) where profiling data seems to always hurt performance, but that is an anomaly. These results suggest that offline profiling conducted over long time intervals may not have much of an advantage over traditional online profiling based JIT compilation systems.

5.0.3 Impact of Profile Data *Accuracy* on Code Quality

AOT compilation systems cannot generate code that is customized to all conceivable program inputs or to all the different program behavior patterns possible at run-time. These systems can still employ static analysis or offline profiling to guide PGOs to derive additional performance gains. However, several issues remain unresolved. In this section we report our observations from a series of experiments conducted to understand two important issues. First, how similar does the guidance provided to the PGOs by the training and evaluation inputs need to be to generate comparable quality code; and second, how much do non-representative program inputs affect quality of guidance provided to PGOs and what is the resulting performance impact.

5.0.3.1 Offline Profiling with DaCapo *small* Input

DaCapo benchmarks are shipped with two input sets for each program, *small* and *default*. In this section, we first evaluate the effectiveness of recording the program behavior with the *small* input, and using that data to guide PGOs during an evaluation run with the *default* input set.

We use the setup described in Sections 4.0.2 and 4.0.3 for these experiments. We compare the run-times from two configurations for each benchmark. The first configuration instantiates

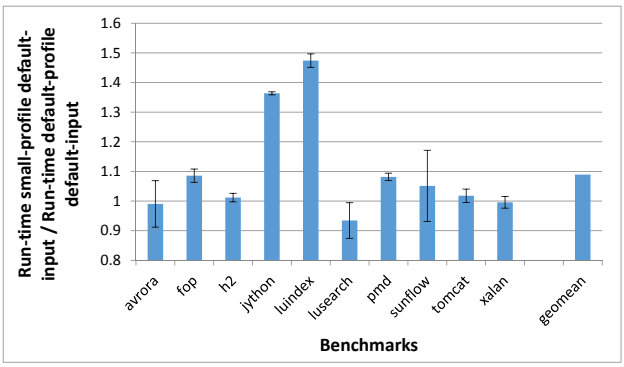

Figure 5.4: In most cases, offline profiling using DaCapo's *small* input produces a binary that achieves good performance with a later evaluation run with the *default* input

the training run for each benchmark with DaCapo's *default* program input. The profile data and method compilation order collected by the training run are exported. The evaluation runs again use the same *default* input size. At the end of the first iteration, the VM stalls all application threads, imports the stored profile data and compiles all the hot methods in the compilation order provided by the training run. The application resumes after all the compilations finish. We allow the program run to stabilize over the next few iterations before recording the benchmark run-time.

The second *offline-profiling* configuration follows a setup very similar to the first. In this case though the training run is instantiated with DaCapo's *small* program input, and it exports the profile data collected. The evaluation run loads the profile data exported by this training run, but uses the method compilation order from the first configuration to compile the same set of hot methods. The program run-time is again recorded at the end of 12 benchmark iterations.

Figure 5.4 displays the run-time reported by the offline-profiling configuration as compared to the run-time from the first configuration for each benchmark. Thus, with the exceptions of *jython* and *luindex*, the *small* input does a very good job of representing the behavior of the program when using the *default* input. On average the performance with offline-profiling only shows a degradation of 8.94% compared to the first configuration that has access to profile data from the same run.

In Section 4.0.4 we described our technique to compare and quantify the *representative-ness* of different program inputs with some given/current program input. Figure 5.5 uses this mechanism to

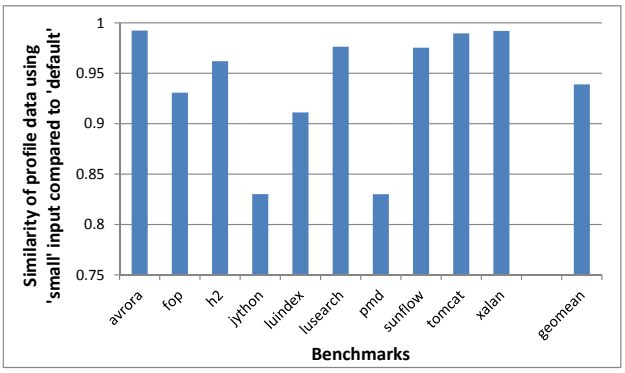

Figure 5.5: Dacapo's *small* input can closely represent the program behavior of a run with DaCapo's *default* input for guiding HotSpot's PGOs.

quantify the similarity or representative-ness of DaCapo's *small* input set compared to the *default* input for each benchmark. We find that with an average similarity score of 94%, program behavior with DaCapo's *small* input is very representative of its behavior with the *default* input, with regards to guiding the PGOs in the HotSpot JVM. This high similarity score explains the offline profiling results observed from Figure 5.4.

5.0.3.2 Offline Profiling with *Randomized* Program Input

Although DaCapo's *small* input set generates a representative profile for the *default* input set for most benchmarks, it is unclear (a) if other programs inputs may provide varying representative-ness, and (b) what is the effect of such plausible variance on the effectiveness of PGOs and the delivered code quality. Unfortunately, we do not know of any popular Java benchmark suite that includes a deliberately and systematically designed diverse set of program inputs. It was also not obvious to us how to generate such input sets for multiple benchmarks that have quantifiably differing similarities. Instead we develop a novel approach to systematically vary the representative-ness of the known program profile for any program-input pair, and study its effect on performance.

Our approach first conducts a training run to collect and export the complete per-method profile data for each benchmark with its standard *default* input set. This profile contains multiple fields, such as branch-taken counts, trap information, etc. Then, we methodically introduce random noise into this profile data with controlled probabilities as each profile data field is loaded during later

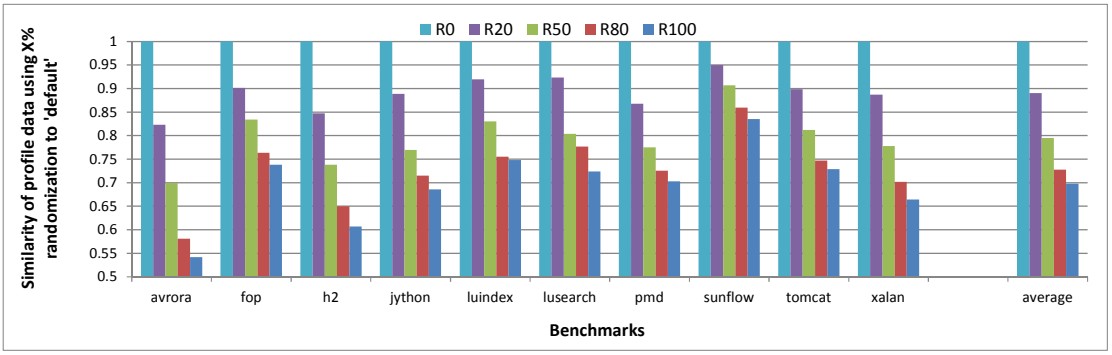

Figure 5.6: Representative-ness of the profile-sites trace with the HotSpot c2 JIT compiler for various randomization configurations and benchmarks as compared to HotSpot's default *reactive* configuration (zero input randomization)

evaluation runs. We call this process *randomization* of the profile data. Thus, a profile data randomization with a probability of X% will alter a profile data field with a probability of X% and leave it unchanged with a probability of (100-X)%.

Randomization of the profile data field depends on the type of the field. For boolean fields, randomization flips the boolean value. For integer counter fields, randomization will set the field to a `low` or `high` value with the same probability. A `low` counter value is guaranteed to be less than the fixed VM threshold for that counter, and a `high` counter value exceeds the threshold. For class pointer fields, a non-`null` field will be set of `null` with the same probability. If the randomization does not nullify the entire field, then each referenced class in that field may again be set to `null` with the same probability. We do not yet attempt to alter a class pointer to instead reference another random class. Likewise, we also do not attempt to update a `null` class pointer to reference some other random program class. This randomized profile data will be used later during the run by the VM to guide PGOs during JIT compilation of the hot program methods.

Our experiment employs randomization values in increments of 10, from 10% to 100%, and uses more finer-grained randomization values between 0% and 10%. First, we employ our mechanism described in Section 4.0.4 to evaluate and quantify the representative-ness of the resulting profile data after randomization. For each benchmark, Figure 5.6 plots this similarity metric between the profile data with varying randomization factors as compared to profile data provided by the *default* input. We can see that profile data representative-ness decreases with increasing

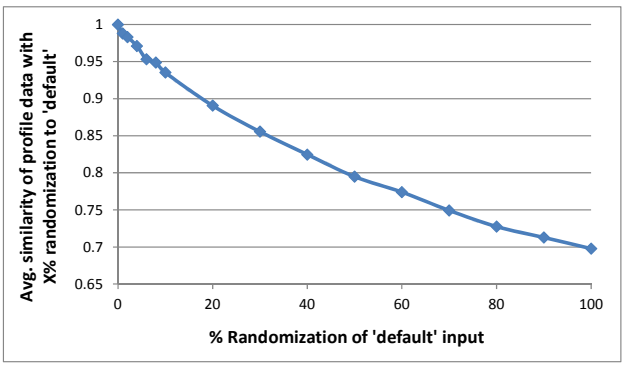

Figure 5.7: Average representative-ness of the profile trace for various randomization configurations as compared to HotSpot's default *reactive* configuration

randomization. While all benchmarks show the same trends, some show a larger scope of similarity metric variation from 0% to 100% than others (for example, *avrora* compared to *sunflow*). Figure 5.7 shows the average representative-ness metric over all our DaCapo benchmarks for all the different randomization ratios attempted. This curve shows that even small profile data imperfections, starting with 1%, noticeably affect the similarity metric. Yet, even a completely random (100% randomization) program input still achieves a similarity metric close to 70%, indicating that even vastly different profiles result in the compiler making similar decisions in a majority of the cases.

Figures 5.8 and 5.9 show the performance implications of using varying levels of imperfect profile data to guide PGOs in HotSpot's c2 compiler. For each benchmark, each bar in Figure 5.8 plots the ratio of program run-time when the VM is using the indicated randomization of profile data to program run-time in the default scenario when using online profile data from the same run with no randomization. Again, we employ the frameworks described earlier in Sections 4.0.2 and 4.0.3 to produce a fair comparison.

These figures provide some interesting observations. All benchmarks show an identical trend with performance degrading with increasing profile data imperfection. However, the scale of the performance change varies significantly between the programs. We also notice that the magnitude of the performance degradation does not show a direct correlation with scale of deterioration in the representative-ness metric. For instance, *sunflow* notices the least deviation in the similarity metric

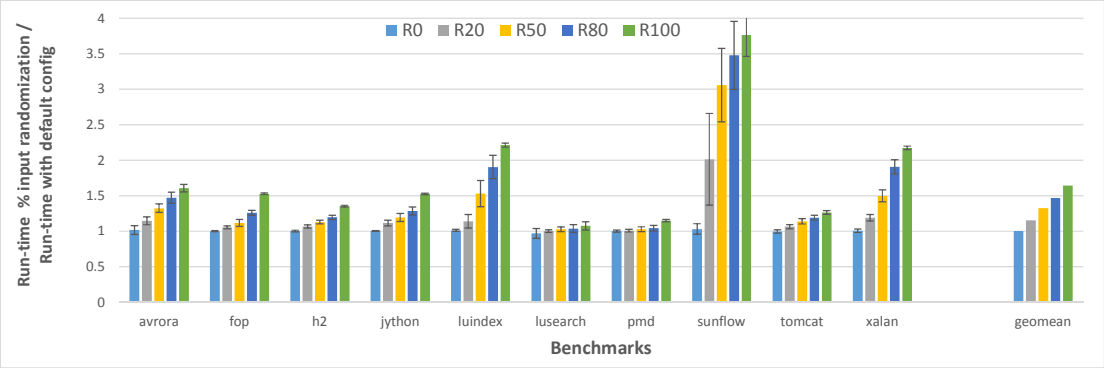

Figure 5.8: Impact of varying profile data inaccuracy on *individual* program performance on the HotSpot JVM

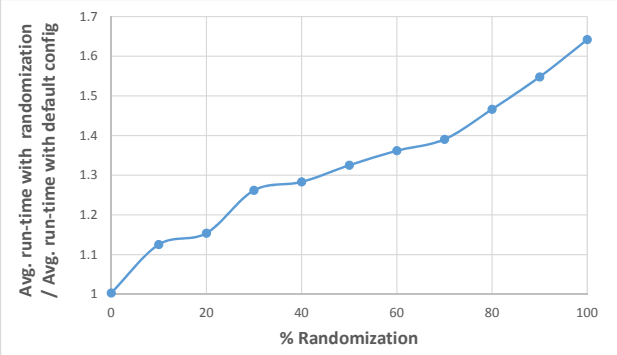

Figure 5.9: Impact of varying profile data inaccuracy on *average* program performance for the DaCapo benchmarks on the HotSpot JVM

from 0% to 100%, but experiences the highest performance variation. Our results show that even small imperfections in profile data can significantly lower the effectiveness of PGOs, which bears important implications for offline profiling based optimization strategies.

5.0.3.3 Randomizing Profile-Site Decisions During Compilation

In the last section we presented results from an experiment that attempted to understand the effect of offline profiling by orderly introducing imperfections into the profile data used to guide PGOs. This profile data is used at various *profile-sites* during compilation to affect optimization decisions. As mentioned earlier, we identify 55 static profile-sites in the source code of HotSpot's c2 compiler where some profile data determines the path taken by the compiler. In this section we study and quantify the sensitivity of the compiler to incorrect decisions taken at profile-sites. Again, we

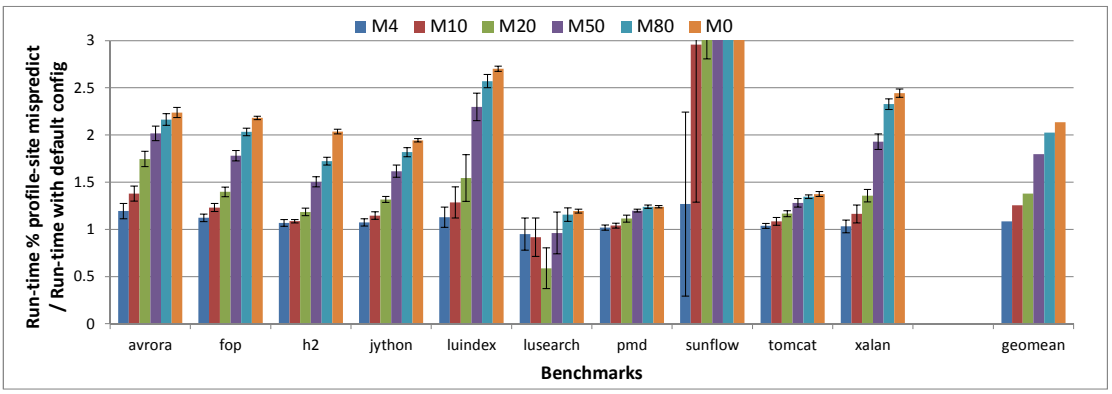

Figure 5.10: Increasing the probability of mispredicting at a profile-site branch increases the negative impact on the quality of generated code on the HotSpot JVM (individual benchmark view)

define accurate profile decisions as those induced by *online* profiling, where the profile data for the current run is dynamically collected by the VM during the same measured program run.

Our experiment to quantify the performance impact of compiler sensitivity systematically varies the probability of the compiler taking a wrong decision (relative to that taken by the online profiling based *reactive* HotSpot configuration) at a profile-site. Our experiment reverses the path taken at each profile-site with a given user-specified probability (referred to later as the *mispredict* probability). Thus, a mispredict probability of 0% forces the c2 compiler to take the same path as that taken by the *reactive* HotSpot configuration every time and at every profile-site reached during compilation. In contrast, a mispredict probability of 100% forces the c2 compiler to take the *wrong* path at every profile-site, whenever feasible. [1]

Figures 5.10 and 5.11 show the impact of different mispredict probabilities on program performance as compared to the program run-time achieved by the default reactive HotSpot configuration with 0% mispredict probability. Thus, we find that even a small mispredict probability causes a noticeable degradation in generated code quality. A mispredict probability of 4% increases program run-time by 8.5%, while 10% misprediction causes a large 25% slowdown. Thus, our experiments show that the HotSpot c2 compiler relies on most/all profile-sites to be correctly predicted to max-

[1] It is not always feasible to take the wrong path. For instance, if the profile-site references a profile data type that is a class pointer, and the profile data recorded by the *reactive* configuration is `null`, then taking the reverse path may require us to now provide an actual plausible class pointer value. Our setup does not yet have the capability to construct such values.

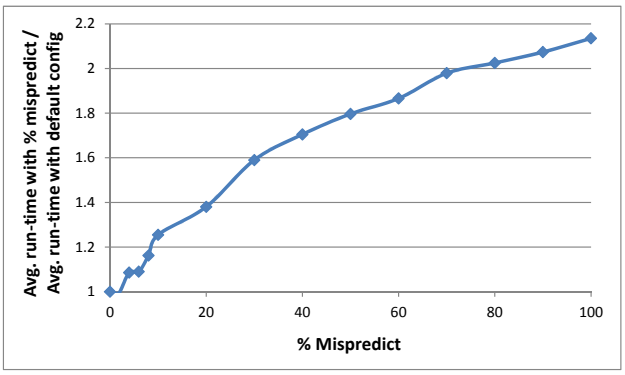

Figure 5.11: Impact of varying the probability of mispredicting at a profile-site branch on *average* program performance for the DaCapo benchmarks on the HotSpot JVM

imize effectiveness. This result is important to static analysis based prediction techniques that may be employed by AOT compilers to guide PGOs.

5.0.4 Contribution of Different Profile Data Types for Performance

In this section, we investigate the relative importance of different profile data types to performance. If certain profile types are more important to PGO effectiveness, then researchers may be able to focus their efforts to more precisely predict those values using static analysis or other techniques.

We manually studied all the profiling data types used by the HotSpot JVM and categorized them into eight categories. Our eight profile data-type categories are described in Table 5.1. The first column gives the name of each category, and the second column provides a short description of data values that belong to each category. The third column in Table 5.1 gives the number of profile sites that use a data-type from a particular category to make optimization decisions in the compiler.

To quantify the performance impact of each category of profile data-types, we execute each benchmark application with certain sets of profile sites *disabled*. We disable a particular profile site by forcing the compiler to take the path that would be taken if no profile data were available at that point. Otherwise, if the site is enabled, the VM simply uses the online profiling information collected earlier in the same program run. For each profile site category, we execute each benchmark with all of the profile sites disabled except for the sites belonging to that category. Ad-

Name	Description	# sites
All_traps	Determines whether a trap event, such as an array-out-of-bounds exception, occurs at a particular BCI or in a given method.	32
Null_at_BCI	Whether or not a null was observed at a particular BCI. Affects implicit null-check optimizations.	7
Unique_receiver_class	Which subclass is the dynamic type of `this` for a virtual call. Affects type speculation and inlining of virtual calls.	2
Receiver_class_count	The count of receivers for a virtual call, if multiple were observed. Affects type speculation.	1
Klass_for_call	The dynamic types of the arguments and/or return values for a call. Affects type speculation.	3
Inv_loop_counters	Influences "warm-call", or relatively but not absolutely hot call, inlining.	2
Branch_data	Whether or not a particular branch was taken. For `switch` structures, which `case` was taken.	4
Call_site_count	The number of times a particular call was executed. Affects inlining.	2

Table 5.1: Categories of Profile Data Types in the HotSpot VM

ditionally, we run each of the benchmarks with all of the profile sites disabled. The execution time difference between the run with all sites disabled and the run with only a particular category's sites enabled provides an estimate of the usefulness of that category.

Interestingly, we find that six of the eight profile data-type categories have very little to no performance impact for most benchmarks. `Null_seen_BCI` and `Recv_class_count` are the two categories that show a noticeable impact on performance. Figure 5.12 shows the performance

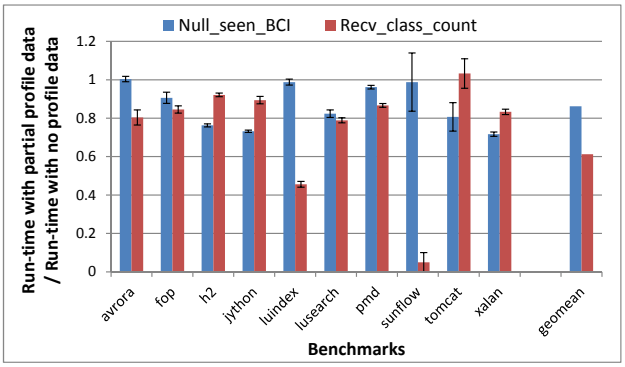

Figure 5.12: Shows the effect of the two profile data types that were observed to impact performance the most.

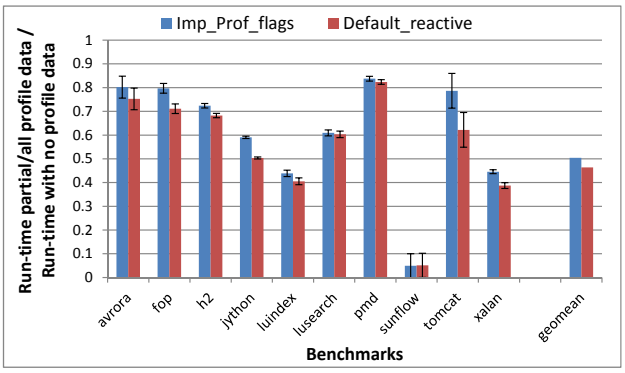

Figure 5.13: Shows the effect of using partial profile data from the six impactful profile data flags, along with the effect of using complete profile data on benchmark performance.

results of this experiment for only these two significant profile data-type categories. Each bar shows the ratio of execution time with a run where only profile sites corresponding to a specific category are enabled to the execution time with all profile sites disabled. Thus, we find that, on average, activating the `Null_seen_BCI` category raises performance by 14%, while activating the `Recv_class_count` category improves performance by 39%, compared to a run that disables all profile-sites.

Data types belonging to the `Null_seen_BCI` category are used at seven profile-sites, while there is only one profile site belonging to the `Recv_class_count` category.

Figure 5.13 shows the result of an experiment that combines the effect of both these categories by only activating the combined eight profile-sites. Additionally, Figure 5.13 plots a separate bar to compare these results to the improvement achieved by a configuration that activates all profiling and all profile-sites. Thus, the comparison reveals that these two categories indeed contribute most of the performance gain due to profiling in HotSpot. This result motivates further research into the effects caused by these two categories of profile data types.

5.0.5 Impact of Generated Code Customization for Different Program Phases

Program execution is known to go through different *phases* [Sherwood & Calder, 1999]. Program behavior in different execution phases vary in some aspects that are important to performance. Therefore, it is conceivable (and perhaps, expected) that some program methods may need to be

customized differently for different program phases. While such customization is feasible in a VM that can perform JIT compilation and detect program phases, it is not as straight-forward to accomplish in an AOT compilation system. Exploiting this issue may further benefit the performance delivered by JIT compilation systems over those employing AOT.

Unfortunately, HotSpot does not yet have the capability to detect program phases, which makes it difficult to evaluate the impact of this issue on performance. We designed an experiment where the training run collects an identical amount of profile information (equivalent to a combined invocation and loop backedge count of 10,000) from different sections of program execution, which is them imported during the evaluation run to guide PGOs for the entire run. The sections selected are method specific and are calculated according to the total execution (invocation + backedge) count of each method. The first section is during the start, the second in the middle, and the last during the end of each method execution.

This experiment did not detect a noticeable performance difference in the three configurations that utilize profile data from different sections of program execution. It is unclear whether this result is because of the absence of phases in the DaCapo benchmarks, or because our blind selection of execution sections did not correctly correspond to actual DaCapo benchmark phases, or because method customization for different phases does not yet produce a significant performance impact in HotSpot. Satisfactorily resolving this issue will need additional VM infrastructure, which we leave as future work.

Chapter 6

Future Work

There are multiple avenues for future work. First, one limitation of this work is that it is only conducted for one VM and compiler, the HotSpot VM's c2 compiler. It is important to investigate if the observations we make in this study can be generalized to other dynamic compilers for Java or other languages. In the future we plan to find other suitable target compilers and perform this generalization. Second, this work demonstrated the benefit that program performance can derive from profile information. At the same time, we also find that the collected profile knowledge needs to be sufficiently detailed and accurate for the current program input for PGOs to realize their maximum potential. Our next research focus will be on how to extract such profile data in systems where online profiling is either not feasible, like AOT systems, and how to customize the generated binaries for different input behaviors. This is a broad research question, with many questions to explore. For instance, similar to categorizing profile data types, can program behaviors also be categorized into a small finite number of behavioral types? Can improvements be made to profile data collection during offline profiling over multiple program inputs so that the dependent optimizations can be specialized to generate variations of binary programs for different behavioral types? Can we build advanced static analysis techniques to improve coverage of offline profiling inputs to encompass all possible program behaviors? Eventually, in the future, we plan to build runtime systems that can combine the advantages of AOT and JIT compilation systems with none, or at least fewer, of the associated drawbacks.

Chapter 7

Conclusion

JIT compilation based VMs can acquire and exploit program profile information from the current run to guide advanced PGOs to generate high-quality native code. AOT compilation systems typically lack access to such reliable profile data, which can restrict its effectiveness. In this work we quantify the impact of profile knowledge on the quality of code produced by JIT optimization systems for dynamic languages like Java. Additionally, we make a number of interesting, and hitherto unknown, discoveries about the properties of profile data that are critical to maximize its ability to correctly guide dependent PGOs. In particular, we find that (a) profile information beyond a small and fixed amount is no longer able to further benefit program performance, (b) small imperfections in profile data can have significant performance implications, (c) a small fraction of profile-site mispredictions can significantly affect the performance of PGOs to generate high-quality code, and (d) although sophisticated VMs, like HotSpot, collect several varieties of profile data, only a few profile data types induce most of the benefits from dependent PGOs. We design and construct several innovative VM frameworks and experiments to accomplish this work. We believe that our frameworks, experiments, and observations can prove useful to VM developers and researchers to build compilation systems that can combine the benefits of both AOT and JIT based models.

References

[Android Open Source Project, 2017] Android Open Source Project (2017). Art and dalvik. https://source.android.com/devices/tech/dalvik/.

[Apple, 1965] Apple, C. T. (1965). Evaluation and performance of computers: The program monitor—a device for program performance measurement. In *Proceedings of the 1965 20th National Conference*, ACM '65 (pp. 66–75). New York, NY, USA: ACM.

[Arnold et al., 2005] Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2005). A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2), 449–466.

[Arnold et al., 2011] Arnold, M., Fink, S., Grove, D., Hind, M., & Sweeney, P. F. (2011). Adaptive optimization in the jalapeno jvm. *SIGPLAN Notices*, 46(4), 65–83.

[Arnold & Grove, 2005] Arnold, M. & Grove, D. (2005). Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the Symposium on Code Generation and Optimization* (pp. 51–62).

[Blackburn et al., 2009] Blackburn, S., Frampton, D., Garner, R., & Zigman, J. (2009). dacapo-9.12-bach release notes. http://dacapobench.org/RELEASE_NOTES.txt.

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., & Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06 (pp. 169–190).: ACM.

[Bowman et al., 2015] Bowman, W. J., Miller, S., St-Amour, V., & Dybvig, R. K. (2015). Profile-guided meta-programming. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15 (pp. 403–412). New York, NY, USA: ACM.

[Chang et al., 1991] Chang, P. P., Mahlke, S. A., & mei W. Hwu, W. (1991). Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21, 1301–1321.

[Cierniak et al., 2000] Cierniak, M., Lueh, G.-Y., & Stichnoth, J. M. (2000). Practicing judo: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00 (pp. 13–26). New York, NY, USA: ACM.

[Duesterwald & Bala, 2000] Duesterwald, E. & Bala, V. (2000). Software profiling for hot path prediction: Less is more. *SIGPLAN Notices*, 35(11), 202–211.

[Georges et al., 2007] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications* (pp. 57–76).

[Github, 2014a] Github (2014a). Dacapo batik benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/1.

[Github, 2014b] Github (2014b). Dacapo eclipse benchmark fails. https://github.com/RedlineResearch/OLD-OpenJDK8/issues/2.

[Graham et al., 1982] Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6), 120–126.

[Hölzle & Ungar, 1996] Hölzle, U. & Ungar, D. (1996). Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4), 355–400.

[Homescu et al., 2013] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., & Franz, M. (2013). Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13 (pp. 1–11). Washington, DC, USA: IEEE Computer Society.

[Hong et al., 2007] Hong, S., Kim, J.-C., Shin, J. W., Moon, S.-M., Oh, H.-S., Lee, J., & Choi, H.-K. (2007). Java client ahead-of-time compiler for embedded systems. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07 (pp. 63–72). New York, NY, USA: ACM.

[Huang et al., 2004] Huang, X., Blackburn, S. M., McKinley, K. S., Moss, J. E. B., Wang, Z., & Cheng, P. (2004). The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04 (pp. 69–80).: ACM.

[Hwu et al., 1993] Hwu, W.-M. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E., Holm, J. G., & Lavery, D. M. (1993). The superblock: An effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7(1-2), 229–248.

[Jantz et al., 2015] Jantz, M. R., Robinson, F. J., Kulkarni, P. A., & Doshi, K. A. (2015). Cross-layer memory management for managed language applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015 (pp. 488–504). New York, NY, USA: ACM.

[Jung et al., 2014] Jung, D.-H., Moon, S.-M., & Oh, H.-S. (2014). Hybrid compilation and optimization for java-based digital tv platforms. *ACM Trans. Embed. Comput. Syst.*, 13(2s), 62:1–62:27.

[Knuth, 1971] Knuth, D. E. (1971). An empirical study of fortran programs. *Software: Practice and Experience*, 1(2), 105–133.

[Krintz et al., 2000] Krintz, C., Grove, D., Sarkar, V., & Calder, B. (2000). Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8), 717–738.

[Kulkarni, 2011] Kulkarni, P. A. (2011). JIT Compilation policy for modern machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11 (pp. 773–788).: ACM.

[Mock et al., 2000] Mock, M., Chambers, C., & Eggers, S. J. (2000). Calpa: A tool for automating selective dynamic compilation. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (pp. 291–302).

[Moseley et al., 2007] Moseley, T., Shye, A., Reddi, V. J., Grunwald, D., & Peri, R. (2007). Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07 (pp. 198–208).: IEEE.

[Mytkowicz et al., 2010] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2010). Evaluating the accuracy of java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10 (pp. 187–197).: ACM.

[Oh et al., 2015] Oh, H.-S., Yeo, J. H., & Moon, S.-M. (2015). Bytecode-to-c ahead-of-time compilation for android dalvik virtual machine. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15 (pp. 1048–1053). San Jose, CA, USA: EDA Consortium.

[Paleczny et al., 2001] Paleczny, M., Vick, C., & Click, C. (2001). The java hotspot^{TM}server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium* (pp. 1–12).: USENIX.

[Pettis & Hansen, 1990] Pettis, K. & Hansen, R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90 (pp. 16–27).

[Robinson et al., 2016] Robinson, F. J., Jantz, M. R., & Kulkarni, P. A. (2016). Code cache management in managed language vms to reduce memory consumption for embedded systems. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016 (pp. 11–20). New York, NY, USA: ACM.

[Rubin et al., 2002] Rubin, S., Bodík, R., & Chilimbi, T. (2002). An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02 (pp. 140–153).: ACM.

[Sherwood & Calder, 1999] Sherwood, T. & Calder, B. (1999). *Time varying behavior of programs*. Technical Report UCSD-CS99-630, UC San Diego.

[Suganuma et al., 2005] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., & Nakatani, T. (2005). Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Transactions on Programming Languages and Systems*, 27(4), 732–785.

[Wang et al., 2011] Wang, C.-S., Perez, G., Chung, Y.-C., Hsu, W.-C., Shih, W.-K., & Hsu, H.-R. (2011). A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11 (pp. 15–24). New York, NY, USA: ACM.

[Wu & Larus, 1994] Wu, Y. & Larus, J. R. (1994). Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27 (pp. 1–11). New York, NY, USA: ACM.